Software benchmarking

http://bench.cr.yp.to

D. J. Bernstein
University of Illinois at Chicago

Joint work with:
Tanja Lange
Technische Universiteit Eindhoven

Selecting cryptographic primitives

NIST's final AES report, 2001:

"Security was the most important factor in the evaluation . . . Rijndael appears to offer an *adequate* security margin. . . . Serpent appears to offer a *high* security margin."

(Emphasis added.)

So why didn't Serpent win?

Selecting cryptographic primitives

NIST's final AES report, 2001:

"Security was the most important factor in the evaluation . . . Rijndael appears to offer an *adequate* security margin. . . . Serpent appears to offer a *high* security margin."

(Emphasis added.)

So why didn't Serpent win?

Maybe hardware efficiency?
Or side-channel security?
Or something else?

Side channels: “The operations used by Serpent are among the easiest to defend against timing and power attacks.”

Side channels: “The operations used by Serpent are among the easiest to defend against timing and power attacks.”

Hardware speed: “Serpent is well suited to restricted-space environments . . . Fully pipelined implementations of Serpent offer the highest throughput of any of the finalists for non-feedback modes. . . . Efficiency is generally very good, and Serpent’s speed is independent of key size.”

Side channels: “The operations used by Serpent are among the easiest to defend against timing and power attacks.”

Hardware speed: “Serpent is well suited to restricted-space environments . . . Fully pipelined implementations of Serpent offer the highest throughput of any of the finalists for non-feedback modes. . . . Efficiency is generally very good, and Serpent’s speed is independent of key size.”

Great! Why didn’t Serpent win?

Aha: Software speed!

Aha: Software speed! “Serpent is generally the slowest of the finalists in software speed for encryption and decryption. . . . Serpent provides consistently low-end performance.”

Aha: Software speed! “Serpent is generally the slowest of the finalists in software speed for encryption and decryption. . . . Serpent provides consistently low-end performance.”

Conclusion: “NIST judged Rijndael to be the best overall algorithm for the AES. Rijndael appears to be consistently a very good performer in both hardware and software [and offers good key agility, low memory, easy defense, fast defense, flexibility, parallelism].”

2007 NIST SHA-3 call: “The security provided by an algorithm is the most important factor in the evaluation.”

2007 NIST SHA-3 call: “The security provided by an algorithm is the most important factor in the evaluation.”

Will this factor alone decide the winner? Perhaps, but I doubt it! Many of the SHA-3 candidates seem extremely hard to break.

Presumably decision will depend partially on speed in software, speed in hardware, speed of implementations with various side-channel defenses, etc.

Speed variability

Main question in this talk:
"How fast is hash software?"

Answer varies from
one hash function to another.

Perhaps this variability
is important to hash users.

Perhaps this variability will be
important in the SHA-3 selection.

Answer depends on
hash-function parameters.

On a 3200MHz AMD
Phenom II X4 955 (100f42),
changing from 256-bit output
to 512-bit output makes
BMW $\approx 1.96\times$ faster;
Blake $\approx 1.05\times$ faster;
SIMD $\approx 1.06\times$ slower;
SHAvite-3 $\approx 1.45\times$ slower;
ECHO $\approx 1.71\times$ slower;
Groestl $\approx 2.06\times$ slower.

Answer depends on #cores used for hashing.

2.4GHz Intel Core 2 Duo E4600 (6fd) has 2 CPU cores operating in parallel.

2.4GHz Intel Core 2 Quad Q6600 (6fb) has 4 CPU cores operating in parallel.
Hash twice as many messages per second!

Standard way to *reduce* this dependence: measure hash time on 1 core.

Warning: Single-core speed
is sometimes better than
speed of 4 cores
handling 4 messages in parallel.
Multiple active cores
can conflict in DRAM access etc.

Warning: Single-core speed$\times$4
is usually better than
speed of 4 cores cooperating
to handle 1 long message.

Warning: These issues
(and more issues coming up)
have different effects
on different hash functions.

Back to the main question:
How fast is hash software?

Answer depends on CPU.

In one second, single-core 1500MHz Intel Pentium 4 (f0a) computes SHA-512 hashes of 9500 4096-byte messages.

In one second, single-core 3192MHz Intel Pentium 4 (f43) computes SHA-512 hashes of 21300 4096-byte messages.

Standard way to *reduce* this dependence: count cycles; i.e., divide #seconds by clock speed.

1500MHz Intel Pentium 4 (f0a): 157924 cycles to hash a 4096-byte message with SHA-512.

3192MHz Intel Pentium 4 (f43): 150128 cycles to hash a 4096-byte message with SHA-512.

Note: Most CPUs have built-in cycle counters; “RDTSC” etc. Cycles are also a natural unit for serious programmers.

Warning: Different CPUs do different amounts of computation in a cycle.

Warning: Different CPUs with different speeds can have the same name.

Warning: Some CPU operations (e.g. DRAM access) do not scale linearly with clock speed.

Warning: A CPU in 64-bit mode is often faster (but sometimes slower!) than the same CPU in 32-bit mode.

4096-byte SHA-512 timings:

53721 cycles: amd64 architecture (64-bit), 3000MHz Intel Core 2 Duo E8400 (1067a).

80640 cycles: x86 architecture (32-bit), 3000MHz Intel Core 2 Duo E8400 (1067a).

155304 cycles: 1900MHz Intel Pentium 4 (f12).

427760 cycles: 333MHz Intel Pentium 2 (652).

1352448 cycles: 416MHz ARM XScale-PXA270 rev 4 (v5l).

4096-byte SHA-256 timings:

64143 cycles: amd64 architecture (64-bit), 3000MHz Intel Core 2 Duo E8400 (1067a).

65241 cycles: x86 architecture (32-bit), 3000MHz Intel Core 2 Duo E8400 (1067a).

142580 cycles: 1900MHz Intel Pentium 4 (f12).

132342 cycles: 333MHz Intel Pentium 2 (652).

160576 cycles: 416MHz ARM XScale-PXA270 rev 4 (v5l).

How fast is hash software?

Answer depends on message length: hashing long message takes more time than hashing short message.

SHA-512 timings on 3200MHz AMD Phenom II X4 955 (100f42):

55915 cycles for 4096 bytes.

29038 cycles for 2048 bytes.

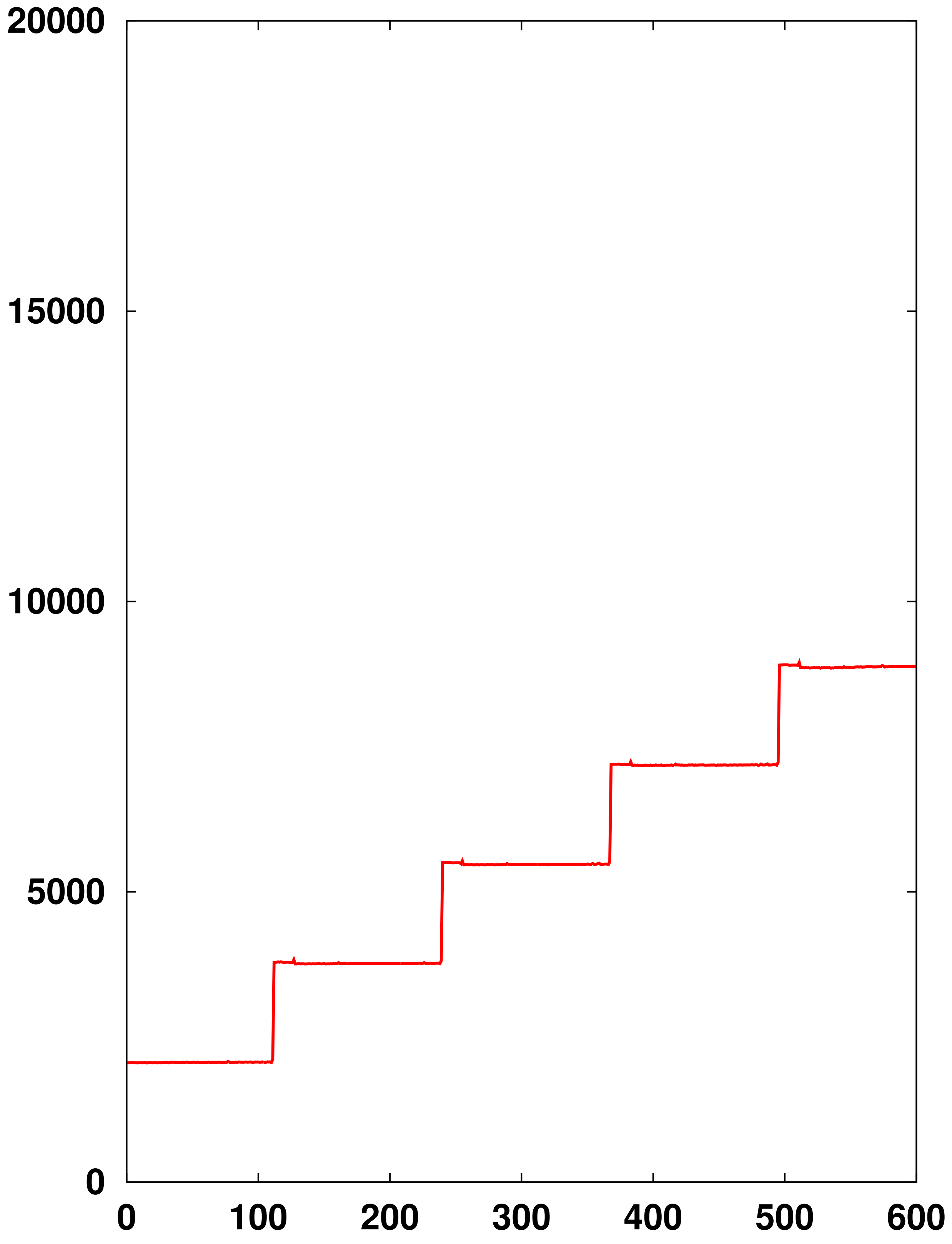
15584 cycles for 1024 bytes.

8861 cycles for 512 bytes.

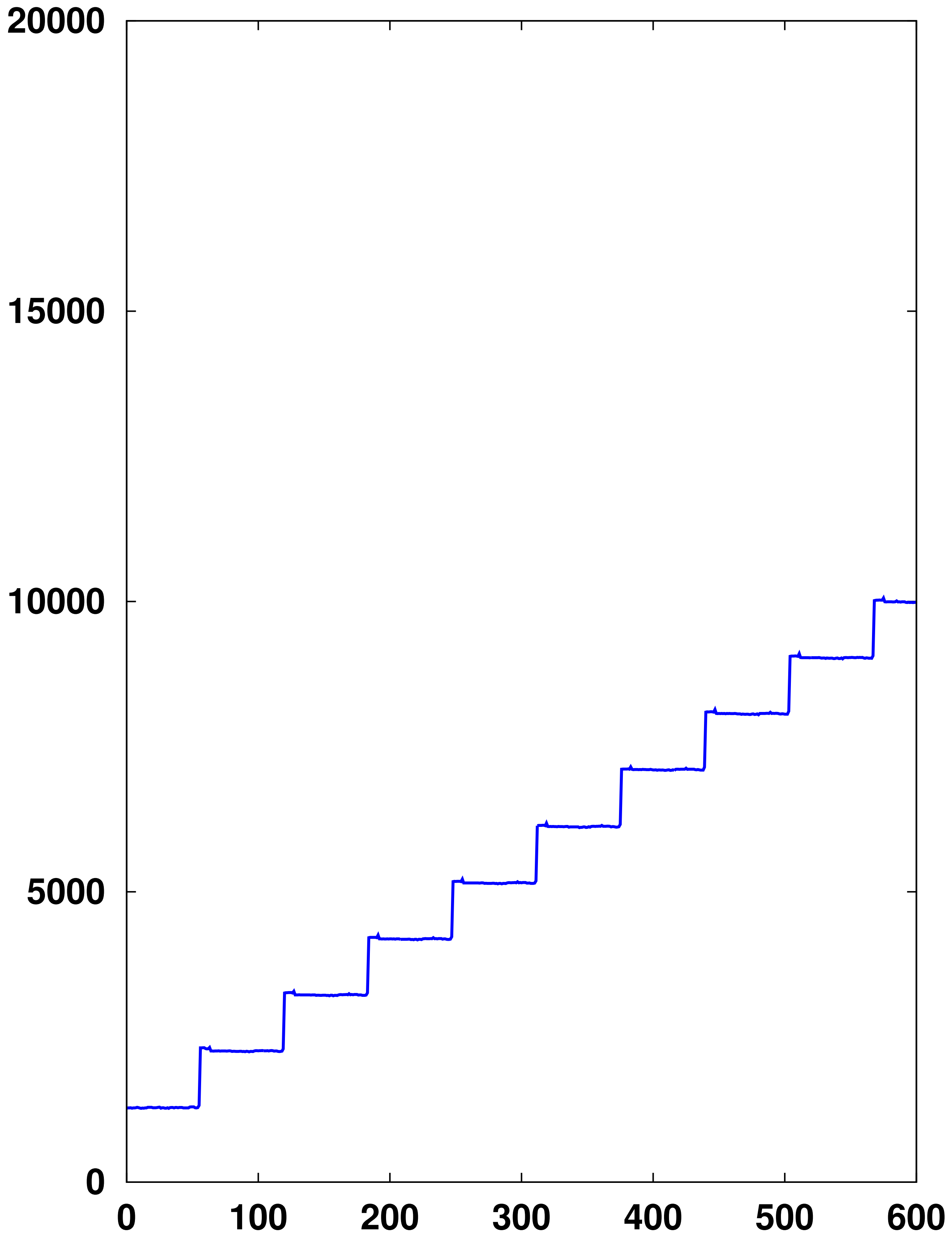
Standard way to
reduce this dependence:
divide cycles by message length.
Warning: Still have dependence.

SHA-512 on the same Phenom:
13.65 cycles/byte for 4096 bytes.
14.18 cycles/byte for 2048 bytes.
15.22 cycles/byte for 1024 bytes.
17.30 cycles/byte for 512 bytes.
21.34 cycles/byte for 256 bytes.
29.35 cycles/byte for 128 bytes.
33.80 cycles/byte for 112 bytes.
19.03 cycles/byte for 111 bytes.
32.15 cycles/byte for 64 bytes.

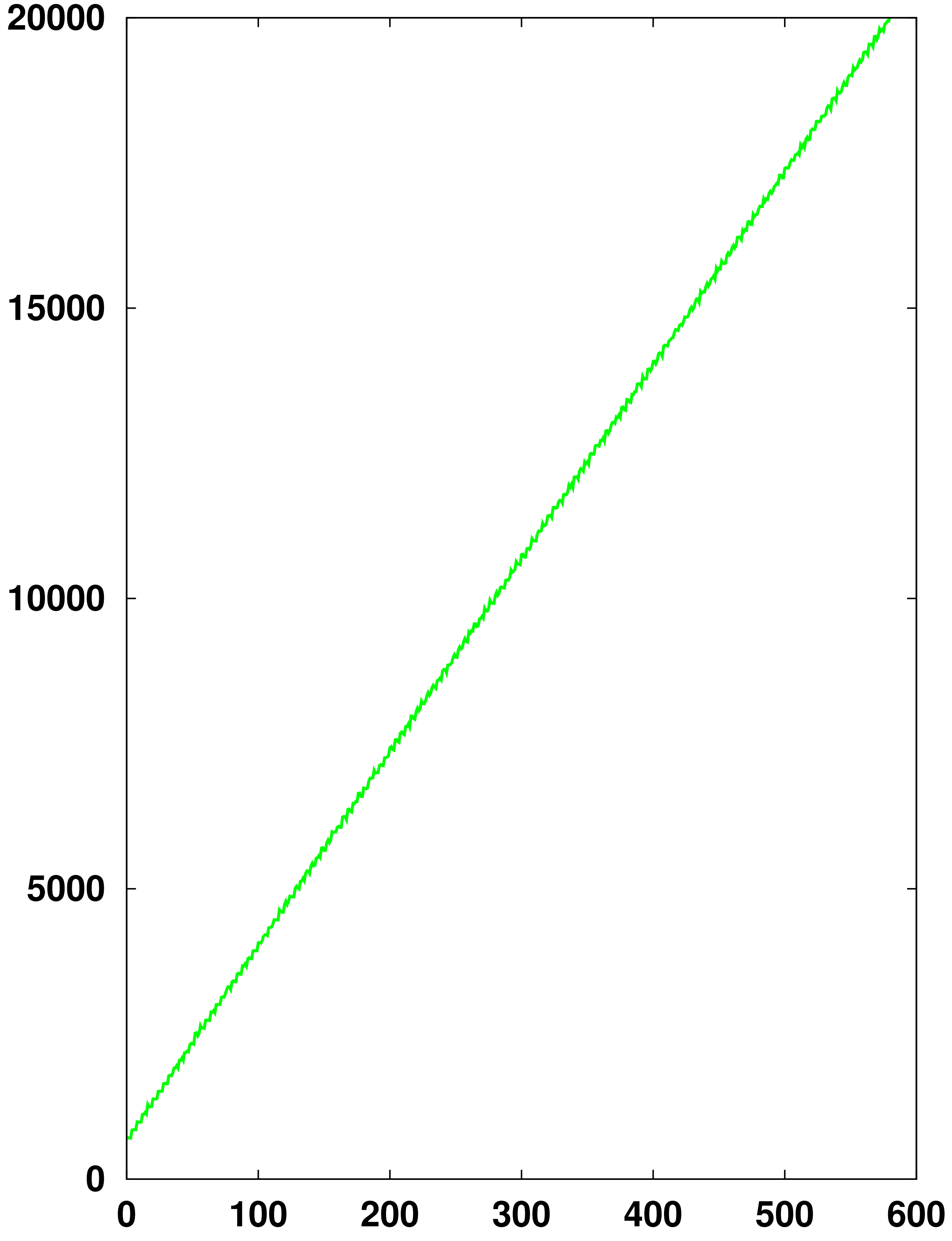
SHA-512 cycles vs. bytes:



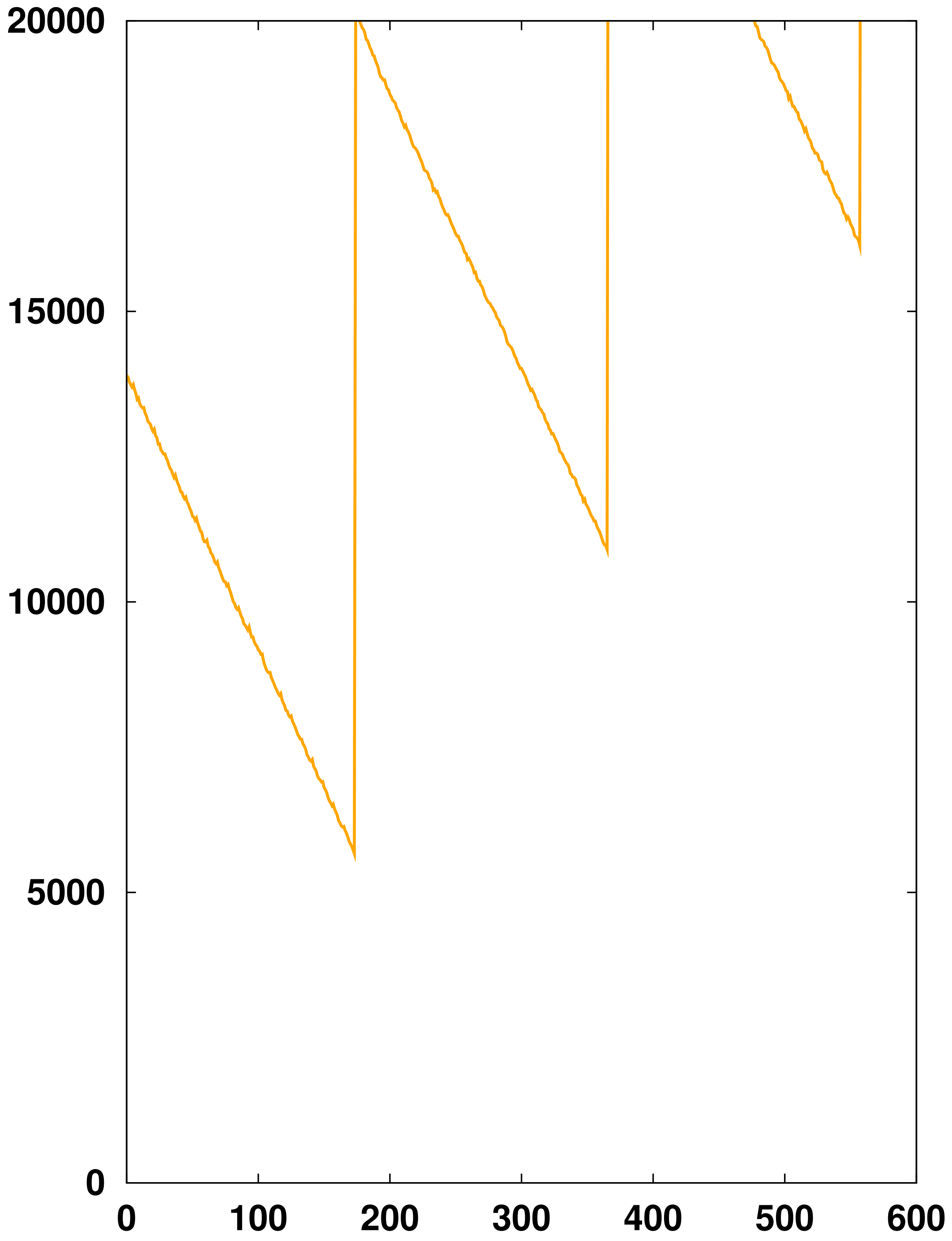
SHA-256 cycles vs. bytes:



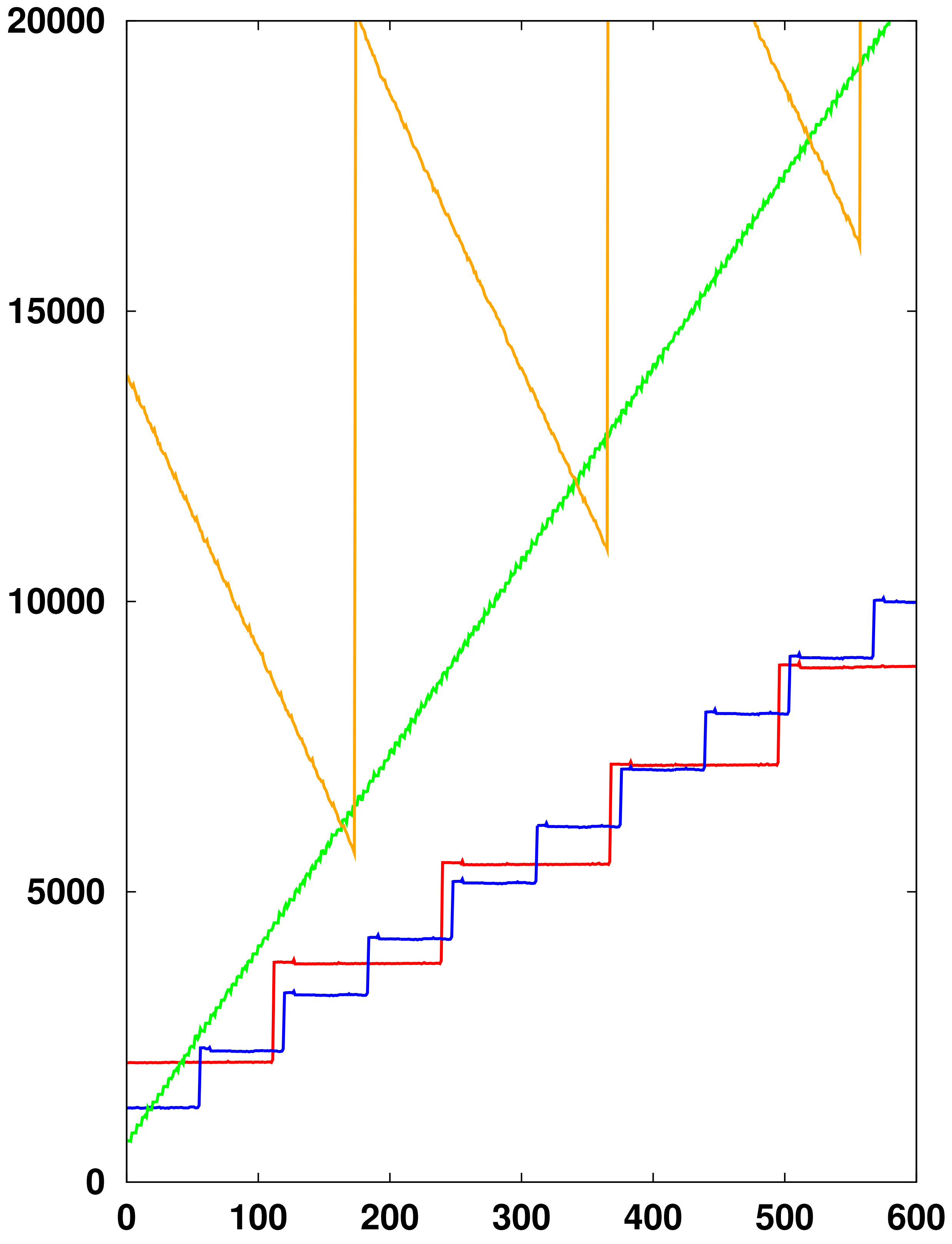
Hamsi cycles vs. bytes:



ECHO-256 cycles vs. bytes:



Cycles vs. bytes:



How fast is hash software?

Answer depends on implementation.

SHA-512: I wrote a simple reference implementation. OpenSSL 0.9.8k is 1.27$\times$ faster than my implementation on a typical Core 2 (for 1536 bytes).

SIMD256: The “vect128” implementation is 5.66$\times$ faster than the “opt” implementation and 247.99$\times$ faster than the “ref” implementation.

A user who cares about speed won't use a slow reference implementation. He'll use the fastest implementation available.

Slowness of unused software has no impact on user's final speed.

The ultimate goal of benchmark reports is to accurately predict the speed that the user will see.
$\Rightarrow$ Report speed of
the fastest implementation.

How fast is hash software?

Answer depends on compiler and on compiler options.

SHAvite-3, Core 2, 1536 bytes:

36824 cycles: `icc -O3 -static`

44840 cycles: `gcc -O -fomit-frame-pointer`

48832 cycles: `gcc -O2 -fomit-frame-pointer`

Warning: There are many other effects on speed.

Answer depends on how much code is in cache.

Answer depends on how many table entries are in cache.

Answer depends on input/output alignment.

Answer depends on bytes being hashed.

Answer depends on details of timing mechanism.

Benchmarking in the dark ages

"I've finally finished
my SANDstorm implementation!
Hmmm, how fast is it?"

Benchmarking in the dark ages

"I've finally finished
my SANDstorm implementation!
Hmmm, how fast is it?"

Traditional answer:
"I'll write a timing tool!
I'll check the clock,
10000$\times$ hash 256 bytes,
check the clock again,
subtract, divide by 10000."

Benchmarking in the dark ages

"I've finally finished
my SANDstorm implementation!
Hmmm, how fast is it?"

Traditional answer:
"I'll write a timing tool!
I'll check the clock,
10000$\times$ hash 256 bytes,
check the clock again,
subtract, divide by 10000."

Maybe more measurements:
"Oops, lots of overhead
in hashing 256 bytes.
I'll try 4096 bytes."

“Okay, 36.6 cycles/byte
for SANDstorm-256
on my 64-bit machine.
NIST says I have to beat SHA-2.
How fast is SHA-2?”

"Okay, 36.6 cycles/byte
for SANDstorm-256
on my 64-bit machine.
NIST says I have to beat SHA-2.
How fast is SHA-2?"

Traditional answer:
"I've written a SHA-256
implementation too.
Let's see . . . 39.1 cycles/byte.
SANDstorm is faster!
This is a *fair comparison*, because
I wrote both implementations,
and put similar effort into both,
and measured both of them
with my own timing tool."

Reality: This SHA-256 software
is embarrassingly slow.
SHA-256 users actually see
much better performance.

To the SANDstorm designer:
You think that SANDstorm can
be made faster too? Prove it!
There's nothing "unfair" about
comparing best available code.

If SANDstorm *can't* run quickly:
comparing lazy implementations
makes SANDstorm look better
than it actually is. Do we want to
reward slow functions? Stupid!

Every dark-ages implementor
builds his own timing tool.
Reports output as “Results”
in an implementation paper.

Summary:
Cryptographic implementor
is the benchmark implementor,
the benchmark operator, and
the competition’s misimplementor.

Every dark-ages implementor
builds his own timing tool.
Reports output as “Results”
in an implementation paper.

Summary:
Cryptographic implementor
is the benchmark implementor,
the benchmark operator, and
the competition’s misimplementor.

This pattern repeats for
every cryptographic implementor.
Hundreds (thousands?) of
separate ad-hoc timing tools
run on various hardware.

Moving out of the dark ages

European Union has funded
NESSIE project (2000–2003),
ECRYPT I network (2004–2008),
ECRYPT II network (2008–2012).

NESSIE's performance evaluators
tuned C implementations
of many cryptographic systems,
all supporting the same API;
wrote a benchmarking toolkit;
ran the toolkit on 25 computers.

Many specific performance results:
e.g., 24 cycles/byte on P4
for 128-bit AES encryption.

ECRYPT I had five “virtual labs.”
STVL, symmetric-techniques lab, included four working groups.
STVL WG 1, stream-cipher group, ran eSTREAM (2004–2008).

De Cannière *published* eSTREAM benchmarking toolkit.

Stream-cipher implementations matching the benchmarking API were contributed by designers, *published*, often tuned; benchmarked on many computers.

e.g. 18 cycles/byte on P4 for third-party asm AES in toolkit.

2006: VAMPIRE, “Virtual Application and Implementation Lab,” started eBATS (“ECRYPT Benchmarking of Asymmetric Systems”), measuring efficiency of public-key encryption, signatures, DH.

Published a new toolkit.

Project is continuing.
Has written, collected, published 55 public-key implementations matching the benchmarking API.
Benchmarked on many computers.

2008: VAMPIRE started eBASC
("ECRYPT Benchmarking
of Stream Ciphers") for
post-eSTREAM benchmarks.

VAMPIRE also started eBASH
("ECRYPT Benchmarking
of All Submitted Hashes").

eBACS ("ECRYPT Benchmarking
of Cryptographic Systems")
includes eBATS, eBASH, eBASC.
Continues under ECRYPT II.

New toolkit, API; coordinated
with CACE library (NaCl).
AES now 14 cycles/byte on P4.

Many advantages of eBACS
over dark-ages benchmarking:

- $>$1000 compiler options.
- $>$100 machine-ABI pairs.
- Many message lengths.
- Very high reliability.
- Public verifiability.
- Real API, not only timing.
- Easy for implementor!

Today have 410 implementations.

Biggest disadvantage:
Report latency is high;
hard to use during development.
. . . but we’re working on this.

eBASH $\rightarrow$ public

eBASH has already collected
180 implementations of
66 hash functions in 30 families.

http://bench.cr.yp.to
/results-hash.html
already shows
measurements on 87 machines;
124 machine-ABI combinations.

Each implementation is
recompiled 1353 times
with various compiler options
to identify best working option
for implementation, machine.

Online tables: medians, quartiles of cycles/byte to hash
8-byte message,
64-byte message,
576-byte message,
1536-byte message,
4096-byte message,
(extrapolated) long message.

Actually have much more data.
e.g. Reports show best options.
e.g. Graphs show medians for 0-byte message, 1-byte message, 2-byte message, 3-byte message, 4-byte message, 5-byte message, . . ., 2048-byte message.

Implementor $\rightarrow$ eBASH

Define output size in `api.h`:

```
#define CRYPTO_BYTES 64
```

Implementor $\rightarrow$ eBASH

Define output size in `api.h`:

```
#define CRYPTO_BYTES 64
```

Define hash function in `hash.c`, e.g. wrapping existing NIST API:

```
#include "crypto_hash.h"
#include "SHA3api_ref.h"
int crypto_hash(
  unsigned char *out,
  const unsigned char *in,
  unsigned long long inlen)
{ Hash(crypto_hash_BYTES*8
       ,in,inlen*8,out);
  return 0; }
```

Send to the mailing list
the URL of a `tar.gz`
with one directory
`crypto_hash/yourhash/ref`
containing `hash.c` etc.

Measurements magically appear!
Much easier than trying
to do your own benchmarks.

More details and options:
`http://bench.cr.yp.to`
`/call-hash.html`

You can implement
someone else's function
and show off how cool you are!

Example:
eBASH includes BLAKE speedups
from Peter Schwabe
and from Samuel Neves.

We'd like to see all second-round
SHA-3 candidates covered by
good implementations.
Contact us for coordination.